HW6

Yuzhen Chen

December 20, 2024

1 Section1: Pix2Pix

- Task 1.1: find_fundamental_matrix

Q1 : find_fundamental_matrix

A1 :

```
    def normalize(points):
        centroid = np.mean(points, axis=0)
        scaled_points = points - centroid

        squared_points = scaled_points ** 2
        sum_of_squares = np.sum(squared_points, axis=1)
        sqrt_sum_of_squares = np.sqrt(sum_of_squares)
        mean_distance = np.mean(sqrt_sum_of_squares)

        const_scale = np.sqrt(2) / mean_distance
        transformation_matrix = np.array([[const_scale, 0, -const_scale * centroid[0]],
                                          [0, const_scale, -const_scale * centroid[1]],
                                          [0, 0, 1]])


        ones_row = np.ones((points.shape[0],))
        vstack_result = np.vstack((points.T, ones_row))

        point_after_norm = np.dot(transformation_matrix, vstack_result )
        return point_after_norm[:2].T, transformation_matrix





def find_fundamental_matrix(shape, pts1, pts2):
    """
    Computes Fundamental Matrix F that relates points in two images by the:

        [u' v' 1] F [u v 1]^T = 0
        or
        l = F [u v 1]^T  -- the epipolar line for point [u v] in image 2
        [u' v' 1] F = l'   -- the epipolar line for point [u' v'] in image 1

    Where (u,v) and (u',v') are the 2D image coordinates of the left and
    the right images respectively.

    Inputs:
    - shape: Tuple containing shape of img1
    - pts1: Numpy array of shape (N,2) giving image coordinates in img1
    - pts2: Numpy array of shape (N,2) giving image coordinates in img2

    Returns:
    - F: Numpy array of shape (3,3) giving the fundamental matrix F
    """

    #This will give you an answer you can compare with
    #Your answer should match closely once you've divided by the last entry
    FOpenCV, _ = cv2.findFundamentalMat(pts1, pts2, cv2.FM_8POINT)

    F = np.eye(3)
```

```
    ########################################################################
    # TODO: Your code here                                                 #
    ########################################################################

    norm_pt1, T1 = normalize(pts1)
    norm_pt2, T2 = normalize(pts2)
    # Construct matrix A for the equation Ax = 0
    A = np.zeros((len(pts1), 9))
    for i in range(len(pts1)):

        u1, v1 = norm_pt1[i]
        u2, v2 = norm_pt2[i]
        A[i] = [u2*u1, u2*v1, u2, v2*u1, v2*v1, v2, u1, v1, 1]


    _, _, V = np.linalg.svd(A)
    F = V[-1].reshape(3, 3)
    U, S, V = np.linalg.svd(F)
    S[-1] = 0  # Set smallest singular value to zero
    temp_F = U @ np.diag(S) @ V
    F = T2.T @ temp_F @ T1

    ########################################################################
    #                             END OF YOUR CODE                         #
    ########################################################################
    return F

```

- Task 1.2: compute_epipoles

  Q1 : Please set up G_optimizer and D_optimizer in the train function.

  A1 :

```
            def compute_epipoles(F):
    """
    Given a Fundamental Matrix F, return the epipoles represented in
    homogeneous coordinates.

    Check: e2@F and F@e1 should be close to [0,0,0]

    Inputs:
    - F: the fundamental matrix

    Return:
    - e1: the epipole for image 1 in homogeneous coordinates
    - e2: the epipole for image 2 in homogeneous coordinates
    """
    ########################################################################
    # TODO: Your code here                                                 #
    ########################################################################
    # Compute the singular value decomposition of F
    U, S, V = np.linalg.svd(F)

    # Extract the last column of V to get the right singular vector
    e1 = V[-1]

    # Compute the singular value decomposition of F transpose to get the left singular vector
    U, S, V = np.linalg.svd(F.T)

    # Extract the last column of V to get the left singular vector
    e2 = V[-1]

    # Normalize the epipoles
    e1 = e1 / e1[2]
    e2 = e2 / e2[2]

    ########################################################################
    #                             END OF YOUR CODE                         #
    ########################################################################

    return e1, e2

```

Q2 : Implement the code for the function train as instructed by the notebook.

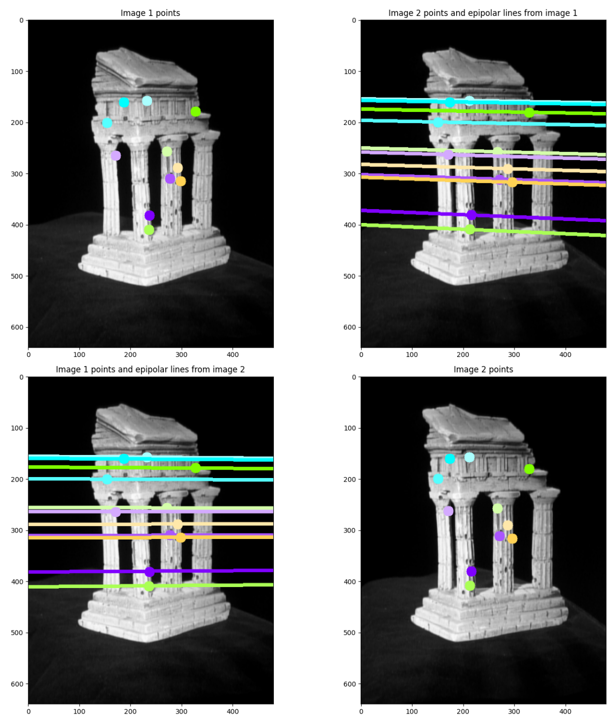
A2 :

- Task 1.3: Show epipolar lines for temple, reallyInwards, and another dataset of your choice.

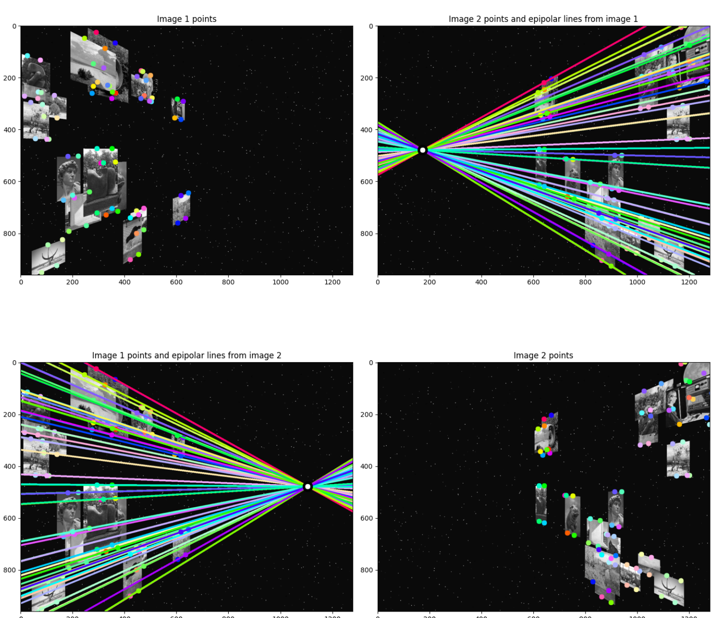
  Q :Show epipolar lines for temple, reallyInwards, and another dataset of your choice.

  A :

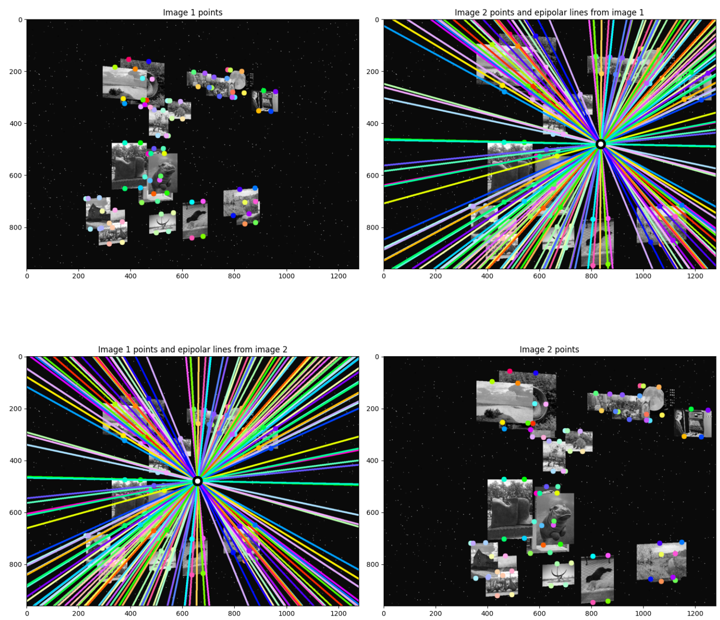
  temple:



  reallyInwards:



  zrtransrot_us:



- Task 1.4: Report the epipoles for reallyInwards and xtrans.

  Q :reallyInwards and xtrans.

  A :
  reallyInwards

```
[1.10594677e+03 4.79305073e+02 1.00000000e+00]
[173.14934439 479.41457107   1.        ]
[1105.94676885  479.30507302] [173.14934439 479.41457107]
```

  xtrans

```
[-8.71977523e+17  2.02228728e+01  1.00000000e+00]
[-3.66017951e+17  3.93111988e+02  1.00000000e+00]
[-8.71977523e+17  2.02228728e+01] [-3.66017951e+17  3.93111988e+02]
```

- Task 2.1 Implement the function positional_encoder(x, L_embed = 6)

```
  def positional_encoder(x, L_embed=6):
"""
This function applies positional encoding to the input tensor. Positional encoding is used in NeRF
to allow the model to learn high-frequency details more effectively. It applies sinusoidal
  functions
at different frequencies to the input.

Parameters:
x (torch.Tensor): The input tensor to be positionally encoded.
L_embed (int): The number of frequency levels to use in the encoding. Defaults to 6.

Returns:
torch.Tensor: The positionally encoded tensor.
"""

# Initialize a list with the input tensor.
rets = [x]

# Loop over the number of frequency levels.
```

```python
  for i in range(L_embed):
    ##############################################################################
    #                                    TODO                                    #
    ##############################################################################
    sin_encoding = torch.sin(2.0 ** i * x)
    cos_encoding = torch.cos(2.0 ** i * x)
    rets.extend([sin_encoding, cos_encoding])
    ##############################################################################
    #                              END OF YOUR CODE                              #
    ##############################################################################


  # Concatenate the original and encoded features along the last dimension.
  return torch.cat(rets, -1)
```

- Task 2.2 Implement the code that samples 3D points along a ray in render

```python
    def render(model, rays_o, rays_d, near, far, n_samples, rand=False):
  """
  Render a scene using a Neural Radiance Field (NeRF) model. This function samples points along rays,
  evaluates the NeRF model at these points, and applies volume rendering techniques to produce an image.

  Parameters:
  model (torch.nn.Module): The NeRF model to be used for rendering.
  rays_o (torch.Tensor): Origins of the rays.
  rays_d (torch.Tensor): Directions of the rays.
  near (float): Near bound for depth sampling along the rays.
  far (float): Far bound for depth sampling along the rays.
  n_samples (int): Number of samples to take along each ray.
  rand (bool): If True, randomize sample depths. Default is False.

  Returns:
  tuple: A tuple containing the RGB map and depth map of the rendered scene.
  """

  # Sample points along each ray, from 'near' to 'far'.
  z = torch.linspace(near, far, n_samples).to(device)
  if rand:
    mids = 0.5 * (z[..., 1:] + z[..., :-1])
    upper = torch.cat([mids, z[..., -1:]], -1)
    lower = torch.cat([z[..., :1], mids], -1)
    t_rand = torch.rand(z.shape).to(device)
    z = lower + (upper - lower) * t_rand

  ##############################################################################
  #                                    TODO                                    #
  ##############################################################################
  # Compute 3D coordinates of the sampled points along the rays.
  points = rays_o[..., None, :] + rays_d[..., None, :] * z[..., :, None]
  ##############################################################################
  #                              END OF YOUR CODE                              #
  ##############################################################################

  # Flatten the points and apply positional encoding.
  flat_points = torch.reshape(points, [-1, points.shape[-1]])
  flat_points = positional_encoder(flat_points)

  # Evaluate the model on the encoded points in chunks to manage memory usage.
  chunk = 1024 * 32
  raw = torch.cat([model(flat_points[i:i + chunk]) for i in range(0, flat_points.shape[0], chunk)], 0)
  raw = torch.reshape(raw, list(points.shape[:-1]) + [4])

  # Compute densities (sigmas) and RGB values from the model's output.
  sigma = F.relu(raw[..., 3])
  rgb = torch.sigmoid(raw[..., :3])

  # Perform volume rendering to obtain the weights of each point.
  one_e_10 = torch.tensor([1e10], dtype=rays_o.dtype).to(device)
  dists = torch.cat((z[..., 1:] - z[..., :-1], one_e_10.expand(z[..., :1].shape)), dim=-1)
  alpha = 1. - torch.exp(-sigma * dists)
  weights = alpha * cumprod_exclusive(1. - alpha + 1e-10)

  ##############################################################################
  #                                    TODO                                    #
```

```
    ############################################################################
    # Compute the weighted sum of RGB values along each ray to get the final pixel color.
    rgb_map = torch.sum(weights[..., None] * rgb, dim=-2)

    # Compute the depth map as the weighted sum of sampled depths.
    depth_map = torch.sum(weights * z, dim=-1)
    ############################################################################
    #                             END OF YOUR CODE                             #
    ############################################################################

    return rgb_map, depth_map
```

- Task 2.3 We will also compute, depth_map,

```
def render(model, rays_o, rays_d, near, far, n_samples, rand=False):
    """
    Render a scene using a Neural Radiance Field (NeRF) model. This function samples points along rays,
    evaluates the NeRF model at these points, and applies volume rendering techniques to produce an
      image.

    Parameters:
    model (torch.nn.Module): The NeRF model to be used for rendering.
    rays_o (torch.Tensor): Origins of the rays.
    rays_d (torch.Tensor): Directions of the rays.
    near (float): Near bound for depth sampling along the rays.
    far (float): Far bound for depth sampling along the rays.
    n_samples (int): Number of samples to take along each ray.
    rand (bool): If True, randomize sample depths. Default is False.

    Returns:
    tuple: A tuple containing the RGB map and depth map of the rendered scene.
    """

    # Sample points along each ray, from 'near' to 'far'.
    z = torch.linspace(near, far, n_samples).to(device)
    if rand:
      mids = 0.5 * (z[..., 1:] + z[..., :-1])
      upper = torch.cat([mids, z[..., -1:]], -1)
      lower = torch.cat([z[..., :1], mids], -1)
      t_rand = torch.rand(z.shape).to(device)
      z = lower + (upper - lower) * t_rand

    ############################################################################
    #                                   TODO                                   #
    ############################################################################
    # Compute 3D coordinates of the sampled points along the rays.
    points = rays_o[..., None, :] + rays_d[..., None, :] * z[..., :, None]
    ############################################################################
    #                             END OF YOUR CODE                             #
    ############################################################################

    # Flatten the points and apply positional encoding.
    flat_points = torch.reshape(points, [-1, points.shape[-1]])
    flat_points = positional_encoder(flat_points)

    # Evaluate the model on the encoded points in chunks to manage memory usage.
    chunk = 1024 * 32
    raw = torch.cat([model(flat_points[i:i + chunk]) for i in range(0, flat_points.shape[0], chunk)],
      0)
    raw = torch.reshape(raw, list(points.shape[:-1]) + [4])

    # Compute densities (sigmas) and RGB values from the model's output.
    sigma = F.relu(raw[..., 3])
    rgb = torch.sigmoid(raw[..., :3])

    # Perform volume rendering to obtain the weights of each point.
    one_e_10 = torch.tensor([1e10], dtype=rays_o.dtype).to(device)
    dists = torch.cat((z[..., 1:] - z[..., :-1], one_e_10.expand(z[..., :1].shape)), dim=-1)
    alpha = 1. - torch.exp(-sigma * dists)
    weights = alpha * cumprod_exclusive(1. - alpha + 1e-10)

    ############################################################################
    #                                   TODO                                   #
    ############################################################################
    # Compute the weighted sum of RGB values along each ray to get the final pixel color.
    rgb_map = torch.sum(weights[..., None] * rgb, dim=-2)
```

```
  # Compute the depth map as the weighted sum of sampled depths.
  depth_map = torch.sum(weights * z, dim=-1)
  ##############################################################################
  #                              END OF YOUR CODE                              #
  ##############################################################################

  return rgb_map, depth_map
```

- Task2.4 Please implement part of the train(model, optimizer, n_iters) function.

```
    mse2psnr = lambda x : -10. * torch.log(x) / torch.log(torch.Tensor([10.])).to(device)

def train(model, optimizer, n_iters=3000):
  """
  Train the Neural Radiance Field (NeRF) model. This function performs training over a specified number of iterations,
  updating the model parameters to minimize the difference between rendered and actual images.

  Parameters:
  model (torch.nn.Module): The NeRF model to be trained.
  optimizer (torch.optim.Optimizer): The optimizer used for training the model.
  n_iters (int): The number of iterations to train the model. Default is 3000.
  """

  psnrs = []
  iternums = []

  plot_step = 500
  n_samples = 64   # Number of samples along each ray.

  for i in tqdm(range(n_iters)):
    # Randomly select an image from the dataset and use it as the target for training.
    images_idx = np.random.randint(images.shape[0])
    target = images[images_idx]
    pose = poses[images_idx]


    ##############################################################################
    #                                    TODO                                    #
    ##############################################################################
    # Perform training. Use mse loss for loss calculation and update the model parameter using the optimizer.
    # Hint: focal is defined as a global variable in previous section


    rays_o, rays_d = get_rays(H, W, focal, pose)

    # Render the scene using the NeRF model
    rgb, _ = render(model, rays_o, rays_d, near=2., far=6., n_samples=64)

    # Calculate MSE loss between rendered RGB image and target image
    loss = torch.nn.functional.mse_loss(rgb, target)

    # Clear previous gradients
    optimizer.zero_grad()

    # Backpropagation
    loss.backward()

    # Update model parameters
    optimizer.step()

    ##############################################################################
    #                              END OF YOUR CODE                              #
    ##############################################################################

    if i % plot_step == 0:
      torch.save(model.state_dict(), 'ckpt.pth')
      # Render a test image to evaluate the current model performance.
      with torch.no_grad():
        rays_o, rays_d = get_rays(H, W, focal, testpose)
        rgb, depth = render(model, rays_o, rays_d, near=2., far=6., n_samples=n_samples)
        loss = torch.nn.functional.mse_loss(rgb, testimg)
        # Calculate PSNR for the rendered image.
        psnr = mse2psnr(loss)
```

```python

        psnrs.append(psnr.detach().cpu().numpy())
        iternums.append(i)

        # Plotting the rendered image and PSNR over iterations.
        plt.figure(figsize=(9, 3))

        plt.subplot(131)
        picture = rgb.cpu()  # Copy the rendered image from GPU to CPU.
        plt.imshow(picture)
        plt.title(f'RGB Iter {i}')

        plt.subplot(132)
        picture = depth.cpu() * (rgb.cpu().mean(-1)>1e-2)
        plt.imshow(picture, cmap='gray')
        plt.title(f'Depth Iter {i}')

        plt.subplot(133)
        plt.plot(iternums, psnrs)
        plt.title('PSNR')
        plt.show()
```

- Task2.5 Please include the best picture (after parameter tuning) of your RGB prediction, depth prediction, and groud truth figure for different view points.